ANL/DIS Draft
Ver: 0.4
9/17/2001 8:50 AM

ATLAS – DataBase (ADB) Architecture Design Document

By
G.H. Chisholm[1]
E.D Frank[2]
D.M. Malon[1]
A.C. Schaffer[3]

September 17, 2001

[1.]Decision and Information Sciences and High Energy Physics Divisions of Argonne National Laboratory and
[2.]Enrico Fermi Institute of The University of Chicago and The Computation Institute of The University of Chicago And Argonne National Laboratory
[3.]Laboratoire de l''Accelerateur Lineaire Universite de Paris-Sud

I. User Guide

Glendower: I can call spirits from the vasty deep.
Hotspur: Why, so can I, or so can any man;
But will they come when you do call for them?

William Shakespeare, "King Henry IV. Part I. Act iii. Sc.1."

1. Abstract

An architecture for the event store component of the ATLAS data-handling system is presented. This document focuses on architecture, leaving design to a separate set of documents. The purpose of this document is to describe the concepts and top-level view of the event store to allow evaluation of suitability. It also will guide and govern the detailed designs of the components stipulated by the architecture, ensuring their interoperability and improving their uniformity in appearance and method. Thus, this document provides a language for discussing, evaluating and building the event store.

The document is divided into a Users Guide, which focuses on concepts, and a Reference Guide that gives details needed to design and build a system. This document is aimed at software developers. A document suitable for general ATLAS collaborators will be written in the future. For the moment, Section I, the Users Guide, will have to serve both audiences, but should be accessible.

We try to define terms prior to use, but there is a glossary at the end.

2. Architecture Motif – Extract and Transform

The ATLAS event store problem is more than just the problem of storing and retrieving data from a file. The full problem is that of data management: how can data be cataloged, shared amongst users at a single site or copied between sites? How can one use the system to help plan and manage computing resources allocated to data access? An event store architecture must satisfy a set of requirements, the most important of which are (See[1]):

1. Store and retrieve transient event data.
2. Store event sample definitions (event lists, hereafter called collections).
3. Support data transfer between sites.
4. Support storage/retrieval optimizations, e.g., listing an event in multiple collections without repeated storage costs.
5. The capabilities list can be implemented in several technologies.
6. Data loss must not bias physics results.

To facilitate data transfer and tape based data-stores, the first architectural decision is that an event, once written, may never be modified. Traditional tape-based High Energy Physics data handling systems had this constraint and supported event-content augmentation by forcing the user to write a new event with new or modified data objects. We provide this “read a tape, write a tape” approach as a basic, easy to use, capability and add storage/retrieval optimization via three additional capabilities:

1. **Event sharing –** two collections may refer to the same event, stored only once
2. **Data Sharing –** two events may refer to common data objects, stored only once
3. **Data Clustering –** data objects frequently used as a group may be stored proximally

In traditional systems, an event could be stored in only one format. The three storage optimization capabilities allow one to store events in a multitude of ways, each with different performance advantages. We refer to these as different *storage formats*. For example, an archival storage format might store all parts of each event which are commonly exported close together in the same file. A remote site might separate the analysis related data, reconstruction data and raw data into separate files to optimize the access to analysis data, e.g., AOD and TAG. See Sections 5.4 through 5.9 for additional examples.

Different storage formats do not imply different storage technologies; rather, different storage formats represent different optimization tradeoffs. A requirement on the architecture is that all storage formats must be readable by the same job, requiring no job configuration changes.

The thesis of the architecture presented here is that ATLAS sites have diverse needs and no single optimization choice will work for all purposes. Choices will differ site-to-site and user-to-user within a site. Therefore, we adopt the following theme of *extract and transform* for the ATLAS data-handling architecture. The architecture will be able express many storage formats to satisfy different optimization needs. The Tier 0 facility will be responsible for storing a copy of the data in a format optimized for record keeping and optimized for extraction of data for export to other sites. This simple format is called the *archival storage format*. Extracted data can be transported to a remote site where it can be rewritten, i.e., transformed, to a format optimized for some purpose. Again, storage format does not connote storage technology, but refers to the multitude of ways to apply event sharing, data sharing and clustering.

This design provides traditional data-handling, i.e., "read a tape, write a tape" as the default job behavior. The database-managed file replaces the tape. Compared to traditional systems, extra features provided by the database architecture allow sophisticated data management optimizations. Each site has the freedom to choose between the pros and cons inherent in any optimization; however, it is the responsibility of the database architecture to ensure there is always a path to extract data into a set of files that can be transferred between any sites. The cost of the extraction depends upon the optimizations chosen by the site and may be high; however, it is the responsibility of Tier 0 to maintain an archive in the archival storage format so that extraction is affordable. The database architecture is also responsible for ensuring site to site uniformity by ensuring that all storage formats can be read without reconfiguring a job.

The position taken in this document is that the database architecture provides more than a set of top-level packages, classes, and interactions among them. It also provides a set of concepts and fundamental operations that form a language for describing, analyzing, and maintaining data-handling scenarios and deployments. Specifically, this language can be used to analyze the architecture itself and, later, to design ATLAS installations. To this end, Section 4 explicitly enumerates the capabilities. Section 5 gives examples of how they are applied to several problems. Part II, The Reference Manual (a sketch of a work in progress), enumerates classes, their interfaces, and their interactions.

We return now to a discussion of the basic concepts in the architecture.

3. Data-management Optimizations: Extensions beyond basic read/write

Traditionally, events were written to files that were archived on tape. If an event was selected into multiple physics categories, then it would be written out multiple times. If the event data needed to be modified, e.g., because tracking was re-run with new calibration constants, then the event would be read in, reprocessed, and written out again to a new file, archived on a new tape. These will be the fundamental operations in the ATLAS database.

Although those are the basic capabilities, we wish to add several new capabilities that will reduce storage costs by avoiding data duplication and that will reduce I/O costs by reading only data actually needed. These capabilities, in concert with current database technologies, should provide tools capable of coping with the large ATLAS data volume. The new capabilities stem from four simple use cases that we describe from the users' point of view.

UC1. If I have a set of events and choose to skim off a subsample, save me the cost of writing the events more than once. (Indexed event access. Reduce disk storage costs.)

UC2. Allow me to avoid reading the full event when I only need a portion of the event to make my selection or perform my computation. (Clustering. Reduce I/O and data unpacking costs.)

UC3. If I choose to abridge or summarize the data in an event (DST formation), allow me to navigate from the abridged event back to the fuller data from which it was derived. For example, allow me to select events based upon summary information (perhaps, 4-vectors and PID), locate rare events in the tails of distributions, and then navigate back to the full reconstruction to allow reconstruction and detector studies, perhaps navigating from there back to raw data. (Data sharing.)

UC4. As an extension to item 3, if I choose to reprocess only a portion of an event, say, calorimetry, allow me to write out only the newly produced calorimetry results but avoid rewriting the other portions of the event, e.g., tracking, thus saving I/O costs and disk costs. (Data sharing.)

Although voiced in terms of a physics analyst, these extensions are also the basis for site level optimization and are the high-level tool set for information managers at sites. Event indexing and data sharing reduce storage costs and clustering reduces I/O costs.

It should be mentioned that CDF finds extensions beyond the essential read/write to introduce complexity and trouble that exceeds any gains. On the other hand, BaBar provides all of these and finds most of them essential to its operations. We choose to provide these extensions since a CDF-like deployment could be formed via a subset of the capabilities.

Next we describe support for basic "read a tape, write a tape" plus the extensions for storage and read optimizations. This description follows the left-to-right flow depicted in Figure 1. The architecture makes use of the Athena notion of output streams. Output streams are the Athena mechanism for expressing event selections or physics categorizations of events. For example, selecting candidate events to three data samples requires that the job be configured to have three output streams. The framework architecture specifies how events are selected into each output stream [2, 3]. The output streams will be assigned collection names in the job configuration.

A collection locates written events. Thus, the storage of collections is separate from that of written events. The advantage of separating the selection record from event storage is that an event can be referred to in many collections without being stored multiple times. This is the notion of *Event Sharing*.

Event sharing requires the ability to select events into multiple streams without storing the actual event data multiple times. Streams can be configured to be writing or non-writing. A stream configured for writing will cause an event to be written prior to registering an event in a collection. A stream configured as non-writing will not cause a write, but will register an event previously written, presumably by another stream, into its collection.

Notice that, given a set of output streams, any given event may be selected into all of them, or any subset, perhaps none. Therefore it is possible that, for some events, only non-writing stream(s) accept the event. We have not set a policy for this eventuality. When a policy is chosen, it will be architectural and uniform among implementations.

Having described navigation to events and event sharing between collections, we now describe data sharing between events. As a design choice, we do not allow arbitrarily fine control of data object sharing. In other words, events do not share things object by object. Instead, they share data objects in groups, called *sharing categories*.

Sharing is done by groups for three reasons. First, data are often processed in stages and the output of one stage might naturally be shared with the next stage, e.g., the reconstruction stage might obtain its raw data by sharing. Second, re-reconstruction must produce groups of self-consistent, re-reconstructed objects. It would not make sense to redo the Tile Cell reconstruction without redoing jet processing. Sharing categories help express the groups of objects that are candidates for re-reconstruction. Third, we need to record a token (*History Token*) with the event data that serves to locate relevant history, e.g., calibration constants used. A history token can be held by the event as a whole and then by any sharing category where it indicates an override of the event level token. In the case of re-reconstruction, the override records the new conditions used in the reprocessing. (See Sections 5.4 and 5.5 and below)

So, while one can conceive of sharing data object by object, our view is that sharing in groups naturally reflects the way data are typically handled and provides a concise means of locating history information at an appropriate level of detail. Also, sharing categories are not just the unit of sharing, but also reflect the granularity of control for representing reprocessing operations.

Given this background, we can describe the persistent event design. The persistent event's responsibility is to locate persistent objects, to express sharing and to store the history tokens. Persistent objects are located via an EventNavigationHeader which has a dictionary style interface: data are registered in the header via a key. They key is anticipated to correspond to the key used to record the transient data in StoreGate. When storing data, you additionally provide a sharing category name to define which sharing category the data will be associated with.

These ideas are illustrated in Figure 2 where two event navigation headers are shown, corresponding to two written events. The event navigation headers locate three distinct groups of persistent objects, i.e., sharing categories, one of which is shared between the two events. The navigation headers point to sharing category headers that define the groups and hold history tokens.

Figure 3 brings all these pieces together. Flowing left to right, it shows two event streams bound to two collections. The collections register event navigation headers that locate persistent objects via sharing category headers.

Everything described so far has been navigational, explaining how events are located, how events can share persistent objects and how collections can share events. Control of object writing is considered next.

In the Athena framework, converters are responsible for converting objects between transient and persistent forms. The database architecture specifies the behaviors of converters and conversion services, uniform across all technologies. These behaviors provide data placement (clustering). We specify them now.

At this time, we have not reached consensus regarding which responsibilities belong to converters and which belong to conversion services. The original presentation left all responsibilities with the converters, and that is how we shall present the architecture here.

Converters write data and we must be able to steer those data to resources. Since an event may be selected into several streams, and because one may wish to put a complete copy of the event into each of the associated collections, a converter may be called more than once for a given event and may need to write to different locations each time: *the storage resource used by a converter is a function of the stream an event is accepted into*.

We also must support storing different pieces of an event into different physical locations in order to support data clustering. For example, the raw data may go to one file, tag data to another and event summary data to a third: *the storage resource used by a converter is a function of the piece of data being written*.

The architecture associates each piece of data in the transient event with one of several placement categories. Placement categories define groups of persistent objects that are to be collocated on disk. This is illustrated in Figure 4.

Figure 4 shows four transient objects, X Y Z and W and their associated converters. The converter (or perhaps conversion service) has been configured so as to categorize the transient objects into placement categories, i.e., has been configured to define groups of collocated objects. Z and X in the figure are identified with placement category one while W and Y are identified with categories two and three. Thus a desire to collocate Z and X has been expressed as has been the desire to separate the storage of Z and X from W and all from Y.

The specific output stream name that has accepted the event is used to lookup storage resources for the various placement categories. This is the role of the Resource Allocation Table (RAT) in Figure 4. In that figure, the output stream that accepted the event (not shown) has storage resources listed in the RAT so that objects steered to placement categories one and two go to a file in the first disk and objects placement category three go to a file in the second disk. If the event depicted was also accepted into another output stream, it is possible for a different set of storage resources to be located for that stream, and therefore the data would be written to different sets of files for the two streams.

Figure 4 indicates that the writing process involves two steps. We just described the writing step, which flows downward in the figure. The second step, which flows to the right, registers the persistent objects in a navigation header so that they can be located in the future. The navigation header is then registered in one collection, or several collections in the case of event sharing.

This completes the description of the extensions beyond the traditional "read a tape, write a tape." The set of self-consistent, fundamental operations required to achieve these extensions allows capabilities (has consequences) beyond the original four. Our view of this situation is as follows. We enumerated four basic capabilities we would like to have and now derive a set of functions to achieve them. You can consider these functions to be like the basic operations in a machine language: from them you can express many ideas. This fundamental set of operations form a language for expressing data-handling systems. That language can be explored to search

for expressible systems that behave unacceptably, even before building them. We can then attempt to modify the set of basic operations to preclude the unacceptable behavior or, failing that, choose to either forgo one of the four capabilities, or choose to not operate some expressible systems that we expect to be problematic.

4. Fundamental Operations

We envision a set of software capabilities plus an archival storage format such that archived data may be transformed into many deployment strategies, each tuned to a suit a particular need. The set of software capabilities fall into 4 genres:

1. Data structural tools.
2. Management-data services (meta data).
3. Concurrency management services.
4. Application interface.

Data structural tools include, for example, a persistent event architecture that supports data sharing between events. Management-data services include, for example, collection lookups by name ("Jet sample") and status ("Good runs") and collection status (resources resident, remote, damaged, etc.). Management-data services also track used database resources and allocates new ones. Concurrency management services include, for example, file-locking.

The 4 genres of capabilities are somewhat abstract. Here we enumerate capabilities at a level more like that in the 9-track tape example. Our purpose here is to challenge the reader to provide examples of essential things that cannot be done, given these capabilities. Another purpose is to provide a set of fundamental operations that can be examined for self-consistency and consequence-predictions.

4.1 Event lookup, filtering and writing:

There are two separate aspects to writing data. The data must be stored and, in addition, navigational information must be updated to allow the data to be retrieved in the future. With regard to navigation, a collection is used to locate a persistent event. The persistent event is used to locate the actual persistent objects. With regard to data storage, placement categories, defined below, are used to steer data to storage resource pools. With this brief introduction, we turn to the capabilities list:

4.1.1 The fundamental access model is that of indexed-sequential access: the event store stores data referred to by persistent events, and indices of persistent events, called collections. Collections allow indexed sequential lookup of events but do not actually hold the events.

4.1.2 You specify a collection, or list of collections, for input.

4.1.3 A read might fetch only a portion of an event. An event may be read even if some data are not resident.

4.1.4 An Athena application can be configured with a list of output streams. Output streams represent selections or categorization of events. An application can select an event into more than one output stream.

4.1.5 Output streams are bound to collections via job configuration.

4.1.6 Output streams can be configured as a non-writing stream, in which case the associated collection is made to index an event that has already been written. Non-writing streams exist to

allow events to be in more than one collection without incurring multiple-writing costs.

4.1.7 Output streams can be configured as writing streams in which case the Athena conversion services will be invoked and the resulting persistent event will be indexed in the collection associated with the writing-stream. One may have multiple writing streams in a single job so that events selected to several streams may all be deep copied to different output collections.

4.1.8 Collections are write-once. They are write-locked upon close, like a file on tape to facilitate data export. (Physics samples grow in time and, therefore, will comprise a list of collections produced by many production jobs. Additional capabilities may be added to assist physics sample management.)

4.1.9 The persistent event's purpose is to allow navigation to persistent objects. Such navigation locates a persistent object by a sharing category name (see next two items) and by a key. The key is usually that used in the transient event for the item being stored. (See Figure 5)

4.1.10 Persistent events are write-once. To modify a persistent event, you must write a new one, either sharing data with the original (see next item) in order to reduce storage costs, or by copying all data.

4.1.11 A persistent event may share data with another persistent event. The granularity of sharing is controlled to limit complexity. The unit of sharing is by groups of objects called sharing categories. (See Figure 2.)

4.1.12 For each transient DataObject to be written or read, there is a converter that converts between transient and persistent forms (data writing) and that modifies the navigation information in the persistent event to allow future retrieval. (See Figure 4.)

4.1.13 Converters are configurable in two dimensions: a) a sharing category for registering the data in the persistent event and b) a placement category to determine where data will be written as described next.

4.1.14 A data-management service has pools of storage resources: there is pool for each placement category. For example a pool for a "Raw" placement category might have a set of files with names like raw00001.dat that will hold just the raw data. See Figure 4.

4.1.15 The placement category for each converter is used to find a storage resource via a resource allocation table that maps the pairs (Placement category, output stream) to storage resource pools. The map includes the output stream so that the same event may be written redundantly to separate physical locations when accepted into multiple streams.

4.2 History tracking:

4.2.1 The persistent event does not store history information, but it facilitates history tracking by storing a history token that may be used by clients to look up history information stored elsewhere.

4.2.2 In addition to a history token associated with the persistent event, a history token may be associated with each sharing category in the persistent event. These express an override for data within that category, relative to the history token for the entire persistent event.

4.3 Resource Tracking and Data Transfer:

We have provided operations that allow events to be selected into physics categories (output streams) and that allow event writing to steer events or pieces of events to one or more storage pools (clustering). To transport data site to site, one must be able to determine what storage resources need to be transported and must be able to copy them and then attach them to the remote site. The capabilities below provide this.

Note that, since the capabilities for event writing allow persistent events to share data, it is possible that storage resources may depend upon each other, e.g., a file may store events that share data with events in another file. Therefore it is possible that attempting to export just a few events require the copy of several large files. This would be an example of where the storage format was optimized for local read access, but not for data transfer. The solution is simple: use the capabilities above to copy the desired events to independent, simply formatted storage resources which then can be exported.

Note also that the prime reason for transporting data is to allow data to be studied elsewhere. Since physics analysis requires one to know the pedigree of data in a sample, one must be able to say where transferred data originated. To maintain data pedigree, there are various rules with regard to freezing below and with regard to write-once in the rules already presented, above.

Now to the capabilities and rules that allow resource tracking and transfer:

4.3.1 A file is the unit of physical storage. It is unspecified whether the file is independent or is part of a database management system. A file holds event data, one or more collections, management data, or pieces of one of these.

4.3.2 You can determine the list of files that store a collection or vice versa.

4.3.3 A file is the unit of data transfer.

4.3.4 Management services allows permanent close (freeze) of a unit of transfer. The freeze is recorded both in management data and in the transfer unit. A freeze is permanent.

4.3.5 A transfer unit must be frozen prior to transport.

4.3.6 A collection is the unit of transfer request.

4.3.7 An Export Program receives a transfer request and produces a transfer control file plus a set of transfer units to transfer.

4.3.8 An Import Program receives a set of transfer units and a transfer control file and installs the new data in the local data handling system.

4.3.9 When files are installed, or created, in a data handling system, they are entered in a file location table which is used by core database code when opening files. Because of this indirection, information managers may move files to effect load balancing or to more effectively utilize tape based hierarchical storage systems.

4.3.10 The Grid collaborates with Management Data services to formulate transfer requests. It achieves a transfer via collaboration with an export program on the export-side to produce a transfer control file and a set of transfer units and via collaboration with an import program which installs the data in the data handling system on the receive side, driven by the transfer control file.

4.4 Resource locking

4.4.1 To be written.

4.5 Authorization and security

4.5.1 To be written.

4.6 Schema management & data evolution

4.6.1 To be written.

5. Fundamental Operations: Examples

In this section we give examples to illustrate the fundamental operations.

5.1 How do I do the simple, traditional "read a tape, write a tape?"

Before, you would have selected a tape volume and file number on that tape to specify the input, then written output to a disk file which later would have migrated to tape (or written directly to a volume/tape positioning number). Now, you do two things.

1. Specify the name of the input collection, or list of collections to the event selector.
2. Specify the name of an output collection.

The default behavior of the system will be to create a default stream for you which will accept all events and that will be bound to your output collection in writing mode. All data will be clustered to land in the same file.

5.2 How do I do the simple, traditional "read a tape, write a tape" but select events into two different outputs?

This is similar to Example 1.

- Specify the name of the input collection, or list of collections.
- Declare two output streams, here called Selection1 and Selection2.
- Associate the streams with decision-makers (you need two) in your job that define the selections.
- Specify the name of an output collection for each output stream.

5.3 How do I filter a collection?

There is a collection I want to filter. I am still exploring the filter options and expect to write many filtered collections as I explore cuts but do not want to waste my disk allocation.

This example is identical to Example 2, but you just need to add an additional configuration. After you declare the output streams instep 2, configure the streams as non-writing streams. The collections you create will refer to the physical events indexed in the input collection(s) rather than writing new ones.

5.4 How do I share data?

I have a collection of events with just raw data. I want to run reconstruction and save the results but do not want to copy the raw data into the output events, but want to save disk by just sharing the raw data between the input and output events. How do I do this?

You do this through additional configuration of the output stream, but there is a pre-requisite. You can only share data at the sharing-category level. Therefore your ability to share with an existing event depends upon how that event was written. In this example, if the first event was written with all of the raw data registered in the same sharing category, say "Raw," then

sharing is possible. In this case, we would configure the output stream for the new job to share "Raw." This declares that navigation header indexed by the collection we are about to write will point to the sharing-category header, keyed under "Raw," that is in the input event.

5.5 How exactly do I get items X, Y and Z into the sharing category called "Trk" while putting W into the sharing category called "Calo?"

In the job configuration, simply configure the converters for X, Y and Z to register their data in the "Trk" sharing category and W in the "Calo" sharing category. As a further illustration, in Example 4, the old job that wrote the raw data would have configured all of its converters to register their data in the "Raw" sharing category.

5.6 How do I cluster data?

How do I get objects of type X to be written to physically adjacent storage, Y to be written to physically adjacent storage, but have X and Y in separate places?

Previous examples addressed data sharing, which is a navigational aspect. This example explores data placement.

?? Choose a placement category for X and a category for Y, say, Tag for X and Aod for Y.

?? Configure the X and Y converters, telling the X converter to use placement category "Tag" and the Y converter to use placement category "Aod."

5.7 Can I steer data when writing without reconfiguring all the converters?

I am a site coordinator and my group wants to rerun the production reconstruction executable on a small sample locally. The production executable is sophisticated and tuned to categorize all the various outputs into half a dozen placement categories. There are 500 converters involved. For simplicity sake, we want all the data to be placed in one file. How do I avoid checking out a dozen packages and reconfiguring 500 converters?

This is one reason why the resource allocation table exists and why converters only categorize their data into a placement category and then use that category to query placement services which then tells the converter where exactly to place the data. In this example, the only action required is to configure the resource allocation table so that the half dozen placement categories all map to the same physical resource pool rather than half a dozen pools. By doing this, although the converters classify the data for placement, you tell the placement services to ignore the categorization and allocate all from the same pool.

This should be the default behavior. Unless told otherwise, the RAT should map all placement categories into the same pool. This simplifies things for the least-trained users, who will be the majority.)

5.8 How do I split apart a set of files containing objects that refer to each other?

I am a site administrator. We imported a large collection of events that passed a loose set of cuts. We want to tighten cuts, separating the collection into two subsamples, but want to force the subsamples into separate files so that we only need to keep on one disk at a time. Note that the imported collection was divided over half a dozen placement categories, and we want to retain this clustering.

This example is quite advanced. This is another example of placement. The job we set up is very similar to Example 2 where events were filtered into two streams. The default behavior, taken there, was to flow all the data out together to a single storage resource. Now what we wish

to do is to use not just two storage resources (two files) but two _sets_ of resources. Each set corresponds to the set of placement categories in the imported data.

The purpose of this example is to emphasize that when converters write their data, their chosen placement category is interpreted through a resource allocation table by the data placement services. This level of indirection is not just a function of the converters placement category, but is _also_ a function of the currently selected output stream. Therefore, two output streams, bound to different collections, can both be configured as write-producing streams, can share (must share!) the same set of converters which categorize the transient data into placement categories the same in both streams, BUT, the final mapping to storage resource (files, roughly) can cause the data to go to two different sets of resources because of this stream dependent redirection in the RAT. To solve the problem in this example you configure two streams, just as in Example 2 and you configure the converters with placement hints to place the data just as it was in the imported data; however, in contrast to Example 2, you edit the resource allocation table configuration so that two different pools of sets of resources are used for the two streams.

5.9 How do I move files from one server to another?

At our site we have found that data in certain collections are used more than others and wish to move those collections onto a faster server and push the other collections onto a slower server. How do I do this?

This example explains capability 4.3.9. To solve this problem, you must use the management data services to obtain the list of files used in the popular collections and in the less used collections. You will move those lists of files onto the various servers according to your load balancing needs. You must then change the file location table and change the locations of these files from their old ones to the new ones.

Another example is when a tape based storage system is involved. If someone requests a file to be loaded from tape, that file might be loaded onto any of a number of disk pools. To avoid the cost of copying the loaded file into a standard location, you can edit the file location table to point to the spool area where the file was loaded.

6. Fundamental Operations: Discussion

We began with the traditional HEP data handling model of read a type, write a tape and then added on four new, basic capabilities to support site-level data management optimization, as described in Section 3. These capabilities added the concepts of event indexing, data sharing and data clustering. In Section 4, we enumerated fundamental operations and rules to achieve these capabilities.

We must ask two questions. Are these capabilities sufficient to satisfy the data handling needs of ATLAS? Are the capabilities few enough and simple enough that we can understand the consequences of their interoperation and demonstrate that the result is supportable and desirable, or are they too complicated? The first question, regarding sufficiency, is somewhat addressed by the examples section, Section 5. This section addresses the second question, regarding unacceptable or unexpected emergent behavior beyond those we sought.

Given the early state of this document, the analysis will not be complete. In fact, it is largely absent; however, we wish to note the need for the analysis and comment on possible responses to problems discovered by the analysis. There are two possible responses.

If behavior emerges that we do not like, one possible response is to remove items from the capabilities list, or add additional rules until the undesired behavior is impossible. One may find

that there is simply no way to provide a desired capability, say data sharing, without introducing the possibility of undesirable behavior. In this first possible response, we would choose to forgo the behavior altogether, reducing the power of the system, e.g., remove data sharing and accept the cost of having to duplicate data on disk.

Alternatively, if behavior emerges that we do not like, we may find that the undesired, emergent behavior only occurs in some, particular applications of the capabilities, or that the problems are problems to some users but features to others. In this case, we may choose to not reduce the capabilities list, but take care to enumerate pitfalls and apply the system in a fitting way for any particular need.

Put simply, what we really do not want are surprises.

Fortunately, the list of capabilities is itself the tool to use to analyze the system. We will work through some use cases in detail regarding files, events sharing data, export scenarios and see what consequences emerge and then decide what to do. We can do this without building the software that implements the capabilities. We turn now to a series of these use cases that we call "Interaction Examples." (This is where the present document requires extension.)

Interaction Example 1: Entanglement via non-writing and writing collections when events are sometimes selected and sometimes not.

Interaction Example 2: Consequences of exporting and importing data when sites have non-identical definitions of resource pools.

Interaction Example 3: Need more…

7. Deployments

get the idea across:

- fundamental operations as tinker toys => multiple constructions you can try to run.
- these are deployments.
- deployment can be good at one task/scenario/site but bad at another or bad everywhere.
- distinguish deployment from architecture.
- should be able to reject deployments without rejecting architecture: tuning and experience.

II. Reference Guide

Note for future revisions: somehow the concept of Domains has been missed in this document and must be addressed in subsequent versions.

1. Introduction

This section attempts more detailed description of how the capabilities in the Users' Guide section are provided. This is an architectural description that enumerates various classes or subsystems, their basic interactions and responsibilities, but does not describe how these agents satisfy their roles. That will be technology dependent and will require further design done in light of this architecture.

Cavaet: This reference guide is incomplete. Only a few class diagrams are presented as placeholders for subsequent development and expansion. Eventually, there will be numerous class, interaction, and activity diagrams that will enforce the architecture on technology dependent implementations. To exercise the premise that the architecture provides sufficient information to effect independent design on disparate pieces of the architecture, two engineers were given this document and asked to say if the architecture seemed feasible in Objectivity (one engineer) and Oracle (the other engineer) and asked to consider specific pieces, like the Database Action Observer (described below). The document has shortcomings, but their feedback was positive and the architecture was evaluated as feasible, perhaps even straightforward.

2. Common Elements

There are a few classes that are shared among the four major subsystems that we will describe here. These classes have few, if any, interesting interrelationships with each other or with other classes.

There are several database entities that store and retrieve things with dictionary-like interfaces; therefore they store and retrieve things by keys. So, we introduce a class called Key that provides operator ==.

A number of entities in the various subsystems have names, for example storage resource pools. We introduce a class called Name that has a method, name(), that returns a String.

For the moment, we have assumed that the simplest choices for these classes will suffice, e.g., that we only need one key. As the design proceeds in coming months, we expect to learn what things to add, e.g., perhaps hashing values for names and keys and order operators for keys.

3. Data Structural Elements

The primary responsibility of the Data Structural Elements is to provide navigation services. The primary duties are

- learning names of collections from a collection catalog,
- navigating from a collection name to an actual collection object,
- navigating from a collection object to a top-level header,
- navigating from that header to actual persistent data items, and
- represent the notion of sharing.

Figure 5 shows the architecture for providing these services. Many of the methods are obvious, e.g., the Collection& lookup(Name) method in CollectionCatalog finds collections based upon their names. We'll concentrate on the less obvious aspects.

The Collection class method getIterator() is motivated by the Athena architecture. In that architecture, event selection services work by producing an Iterator object that can iterate over a source of event data. The Collection class thus says that, if you are writing an event selection service for some technology, you need to write, besides your selection service, a class that inherits from Collection and another that inherits from AthenaBlah::Iterator and then have you event selector locate the Collection via the CollectionCatalog and then obtain an iterator from the collection to return to the framework.

Pushing in deeper, notice that Collections have 0 to N entities in them called EvtNavHdr. These are navigation headers and are the means for locating data with a persistent event. Two points are to be made. First, Collections are sequential indices of EvtNavHdrs. Second, given an EvtNavHdr, you can locate any piece of data associated with that event. We turn to that now.

An Athena-based converter doing a read will locate its data through the EvtNavHdr by using the lookup method that takes two keys; a Key that represents a sharing category and a Key that locates the data. The intention is that the second key should be the same as the key used in the transient event. Similarly, the store method stores a datum, again via those two keys.

The makeShare() method is used to create a sharing category. We do this because we wish to hide the existence of the ShareCategoryHdr, which is discussed below. But the notion of sharing categories exists at the EvtNavHdr level, even if the headers do not, and some level of manipulation of sharing categories is needed. Here, we are saying that it is possible to create new sharing categories for the categorization of data by calling makeShareCat. Note that it is a bool. Further architectural work may allow the system to refuse to create a category in some circumstances. But it shall always be the case that a store will fail if it attempts to store in a category that does not yet exist.

The ShareCategorHdr exists to serve as a navigation collector, of sorts. It exists to enforce the architectural decision that data can not be shared between events with control at the per-object level. Instead, groups of objects are shared, and these groups are called sharing categories. The ShareCategoryHdr represents a sharing group. Therefore, the top level EvtNavHdr does not navigate to the objects directly, but to the ShareCategoryHdr and this is _exactly_ how sharing is implemented: two EvtNavHdr's that share data simply point to the same ShareCategoryHdr.

The ShareCategoryHdr has an interface much like EvtNavHdr. It has lookup and store, but now just on a single key. EvtNavHdr uses the sharing category key to locate a sharing category and then defers the final data by calling lookup in the just-found ShareCategoryHdr.

In short, behaviorally, EvtNavHdr is a dictionary of persistent objects. However, physically, an EvtNavHdr is a dictionary of ShareCategoryHdr's and a ShareCategoryHdr is a dictionary of persistent objects.

The getHistory and setHistory methods store and retrieve the History keys described in the users' guide. Note that the History methods in the ShareCategoryHdr are intended to serve as overrides of the overall key in EvtNavHdr.

Placeholder: explain how non-sequential access mechanisms are to be built in terms of these entities.

Separate documents will describe implementation of this architecture for Objectivity, Oracle and Root.

4. Management Data

Management Data services carries the responsibility for tracking resources used in the database. For example, one must be able to determine what files are used by a collection or what collections (pieces) lie in a file. Although the responsibilities are broad, at this time we only discuss issues related to tracking storage resources at the lowest level.

Figure 6 shows the portion of the Management Data services that handle control of object placement when objects are being written by converters. First we must convey the notion of a storage resource and of a storage resource pool.

Consider a job that is writing many events and assume that one has configured the job to write data into several placement categories, e.g., the placement categories of RAW, TAG and ESD. Think of ESD as "everything else." In this scenario, any given event will be split over three files, one for each of the placement categories. If we are writing many events, those files may fill and we may need to open new ones. Since our objective was to keep the storage of the three components separate, we'd like to monitor the storage resources (files) independently for each of the placement categories. That is the notion of storage resource pools: we have a pool of files for each of the placement categories, e.g., we may choose a naming scheme like tag00001, tag00002, raw00001, etc., within the pools and may store them on different servers.

Although the paragraph above intimates that a storage resource is a file, this may not be the case. For example, in Objectivity, a storage resource would most likely be a container in a database, rather than a file. Thus, the proper way to think of a storage resource is that, given a design within some technology, there will be an intention in that design to control the data writing done by converters. Whatever the unit of control is within the design for that technology, that is what a storage resource would be for that design. Sets of them correspond to storage resource pools.

We now return to Figure 6 and explain the interactions there. Before getting into the details, we first explain the intention. We wish to produce an architecture in which converters can be controlled to steer data into placement categories and in which the placement categories can be mapped onto storage resource pools. What we describe below explains two things. First how converters are controlled. Second, that the names of storage resource pools and placement categories are not hardwired in the system. However, to give data managers control over their systems, valid configurations must specify only resource pools allocated previously.

StorageResource is the base class for all technology-specific storage resource classes. The base class has common methods like hasRoom(), poolName(), etc. Specific derivatives, like ObjyResource, implement these and add technology specific methods like getHint for ObjyResource and getTable() for OracleResource. We will come back to this in a moment.

The class template, PlacementService<T>, is an Athena service. The template Parameter, T, is intended to be a derivative of StorageResource, e.g., one might use PlacementService<ObjyResource>. Converters call T& PlacementService<T>::getResource() (args give placement category ans stream name???) and obtain a strongly typed T. For example, A converter for Objectivity would use PlacementService<ObjyResource>, would call its getResource() method and would get an ObjyResource. Because this is strongly typed, it has

access to the objectiviy specific aspects of the ObjyResource interface, in particular to the ooRef(ooObj) getHint() method.

By virtue of this, converters have access to strongly typed, technology specific storage resource classes. The converter does need to know which placement service to request, but we feel that is natural because the converter is technology specific.

PlacementService also has a method configRAT that is used to configure the resource allocation table. Recall that the RAT maps the pair, (placementCategoryName, StreamName) to a resource pool name. The getResource(Name placementCatName, Name streamName) method works by using the RAT to convert the input names to a pool name and then looking inside the placement service for a StorageResource associated with that pool name. If none is present, then the PlacementService<T> creates a T and stores it under the pool name. If one is present, the PlacementService<T> just returns it to the user. It is subtle, but we have just described a bootstrap. To fully understand what has just happened we must describe the StorageResourceManager<T>.

A StorageResourceManager<T> is responsible for actually implementing policy with regard to storage resources. For example, in Objectivity, it would know how many containers are in each database, how big each must be, what servers hold the databases, etc. This is technology and design dependent. From an architectural point of view, it is responsible for logging information, as needed, to answer queries like, what files are in use, are they frozen, etc. We have not shown here the interfaces for those queries.

A StorageResourceManager<T> is, as for PlacementService<T>, expected to be instantiated only for T's deriving from StorageResource. In the diagram, we show the example of ObjyStorResMgr being a StorageResourceManager instantiated on T being an ObjyResource. We show, schematically for now, methods addResource() and getResource() that manipulate resources in pools. For example, a site-level event-store maintenance program would be written for a given technology that would add new storage resources to an event store. Our intention here is that getResource() can not satisfy requests unless someone has called addResource() and put in resources at some time in the past.

Returning to the diagram, a StorageResourceManager<T> has a map of storage pools, StorageResourePool<T>. When you call addResource(), you will add resources to one of the StorageResourcePools held in the StorageResourceManager<T>'s map. A getResource works the other way around. Why do we have both a StorageResourceManager and a StorageResourcePool? I have no idea. It's a viceral, first guess.

Now we back up again and trace through the chain from the start. A converter wishes to write data of type, say, ObjectVector<Jet>, stored in the transient store with key "KtJetList." It has been configured to use sharing category "Calo" and placement category "ESD." The OutputAgent has determined that the current event is to be saved onto a stream with name, "Stream1," that it is a writing stream and has caused an iteration over converters. The ConversionService has reached our converter for ObjectVect<Jet>.

Our converter uses its configured placement category name and the name of the current stream (we've not worked out how these got into the converter yet). Suppose this is an objectivity converter. It locates PlacementService<ObjyResource> and calls getResource(), passing in the placement category name and stream name. See Figure 7. The PlacementService looks in the RAT and locates a pool name and also an associated ObjyResource which is returned. The converter calls getHint() in ObjyResource which returns an ooRef that it uses to call operator

new to create the persistent object. The converter also uses its sharing category to register the persistent object in the EvtNavHdr.

Suppose the lookup in the RAT had failed which means that no one had yet tried a write to that pool before. Then an ObjyResource would be created with the pool name we wanted to use. Notice that ObjyResource knows how to locate the StorageResourceManager<ObjyResource> which it will defer to to actually know what resource it represents. Also, by virtue of this knowledge, if the ObjyResource is full, it can be updated to refer to a new resource.

This part of the architecture needs to be refined, but the reader should see an interplay of various agents that represent a technology independent description of a system that can steer converters, track resource utilization, and enforce that only allocated resources will be used.

5. Athena Interface

This section will be filled out in time but largely uses existing Athena architecture, e.g., Event selection services, Event conversion services, Output streams.

What needs to be described here is how the capabilities described in the user section relate to configuration of the various Athena agents. We also need to explain the flow of information through the chain.

6. Concurrency Management

Many database management technologies have a, perhaps implicit, state-machine behavior. You must initialize things prior to use, close things at various times, worry about transactions, etc. The conundrum here is that we wish to centralize the policy of when these things happen, but it is many, many pieces of code that do things that might require these state changes. For example, our system will have many converters in it. A converter is a place where the system would know "we are about to write" or "we are only going to read." Nevertheless, we do not want converters, or other objects, to make decisions about this. In particular, we do not want them to start or stop transactions. One good reason for forbidding this is that some technologies may not even have the concept of a transaction. Another reason is that, as policies change in response to system tuning, we have only one place to modify.

We plan to handle this by making a Database Action Observer. The Database Action Observer (Dao) is to be notified of actions taken by code at large. For example, a converter will tell the Dao that it is about to write or is about to read. This is not a request for service! It is simply notification. The programmer writing the at-large database code is to assume there is a stateless system and that the only action on his part is to make the notifications specified in the architectural document.

The Database Action Observer will allow clients to subscribe requests for notifications of actions. The idea is that if a technology requires state maintenance, then it the designers of that system will write a concurrency manager that will subscribe to the Database Action Observer and receive notifications of actions within the code. Because the notifications are of the form "I'm about to do X" and "I've just done X," these notifications can be used as incidents to drive a technology specific state-machine whose actions satisfy the requirements of the technology.

One very simple example of a Database Action Observer client would be a monitor client. The monitor could run in parallel with any technology and measure rates or patterns of actions.

III. End Matter

1. Figures

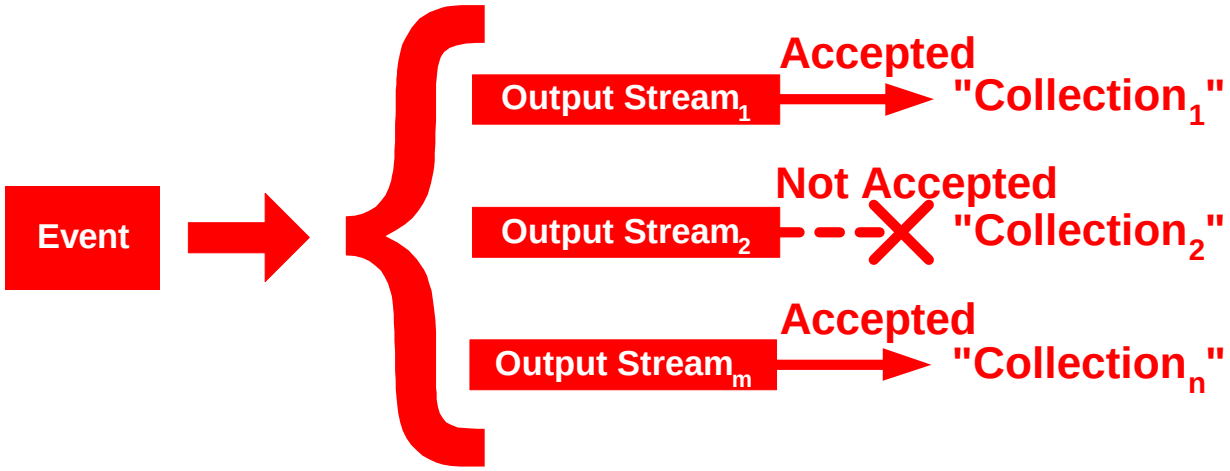

Figure 1: Event Selection into Output Stream(s) and Registration into Collection(s)

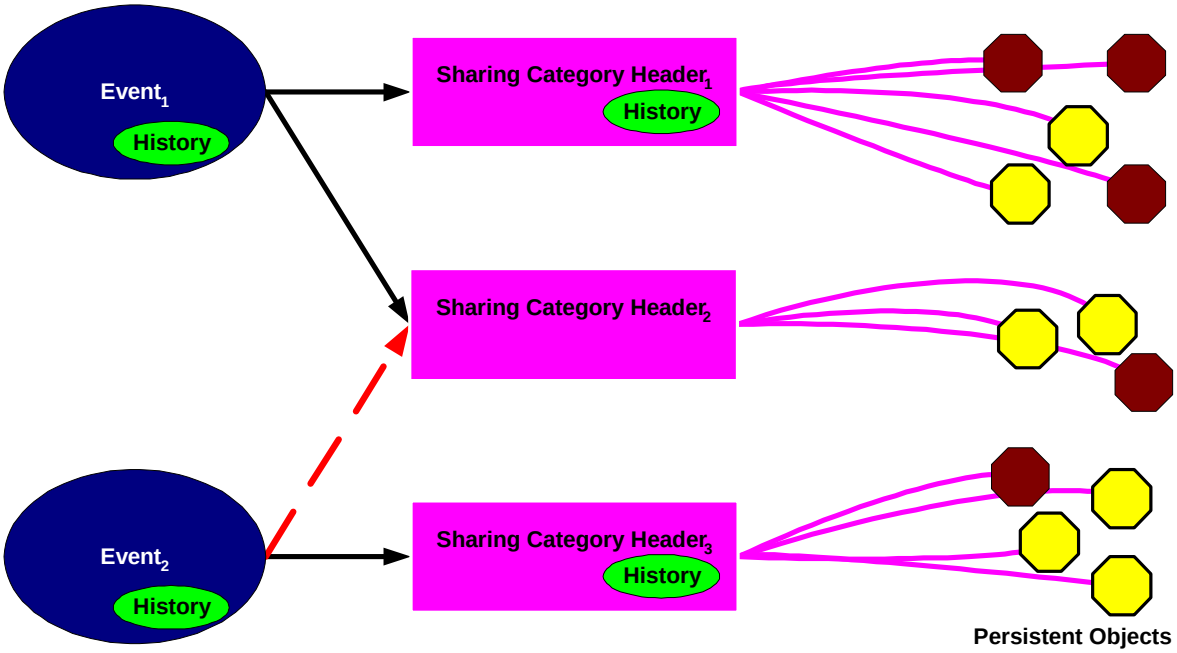

Figure 2: Navigation to persistent objects via Sharing Categories

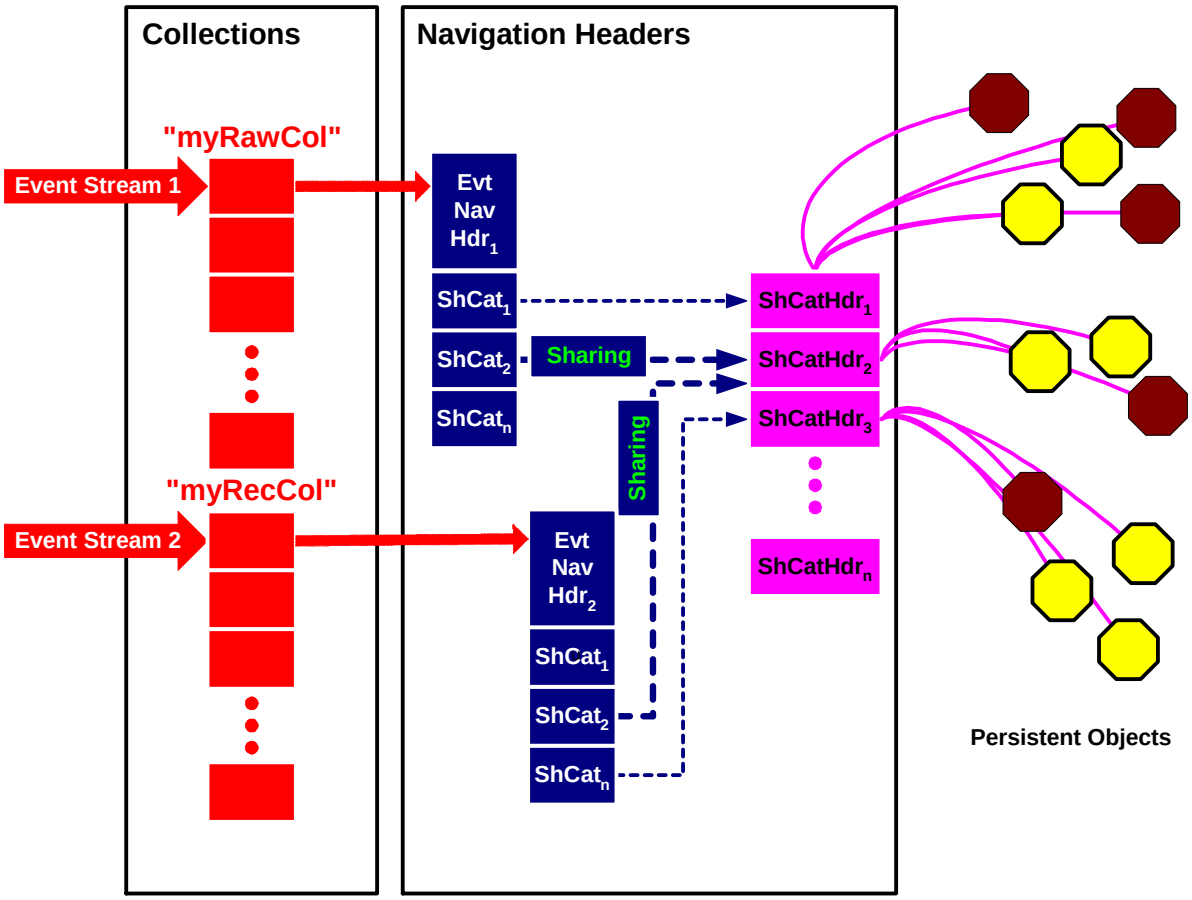

Figure 3: Navigation from Collections to Events to Persistent Objects

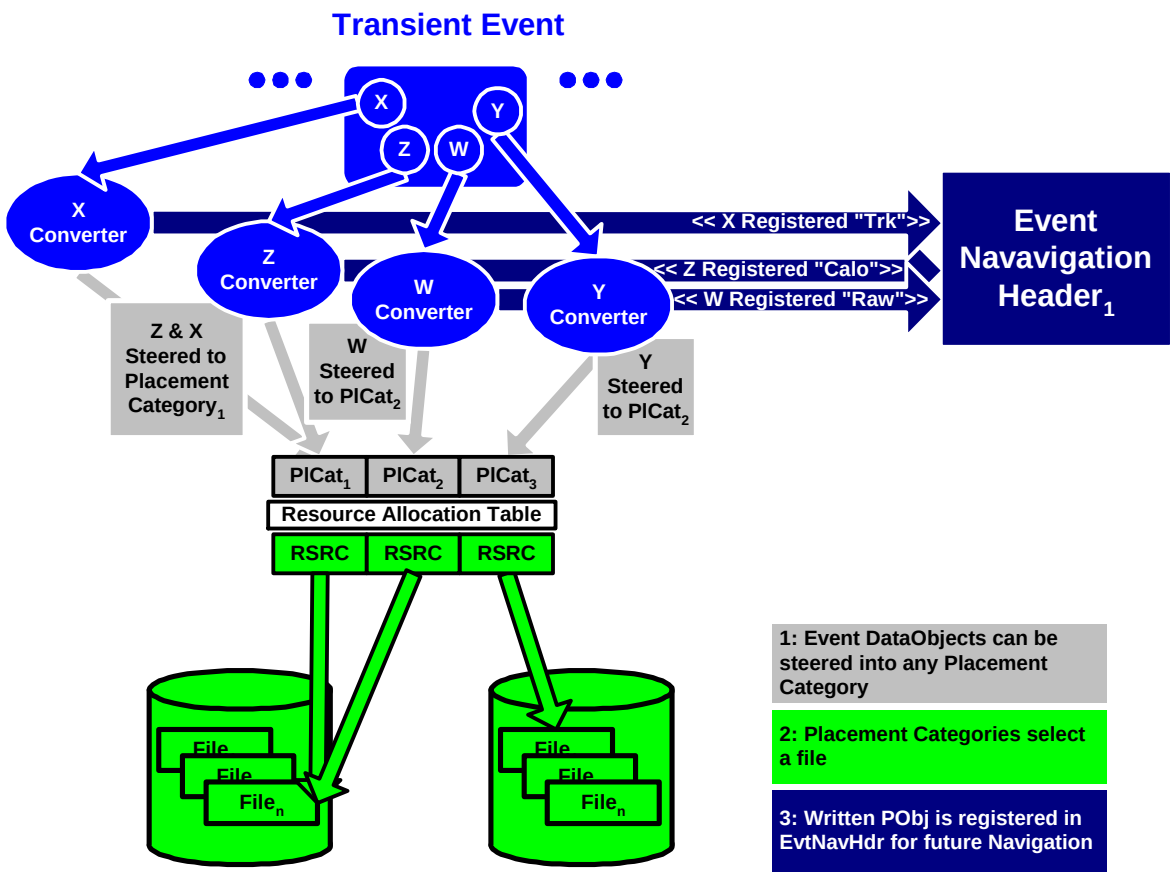

Figure 4: Data Placement (Clustering) and Registration

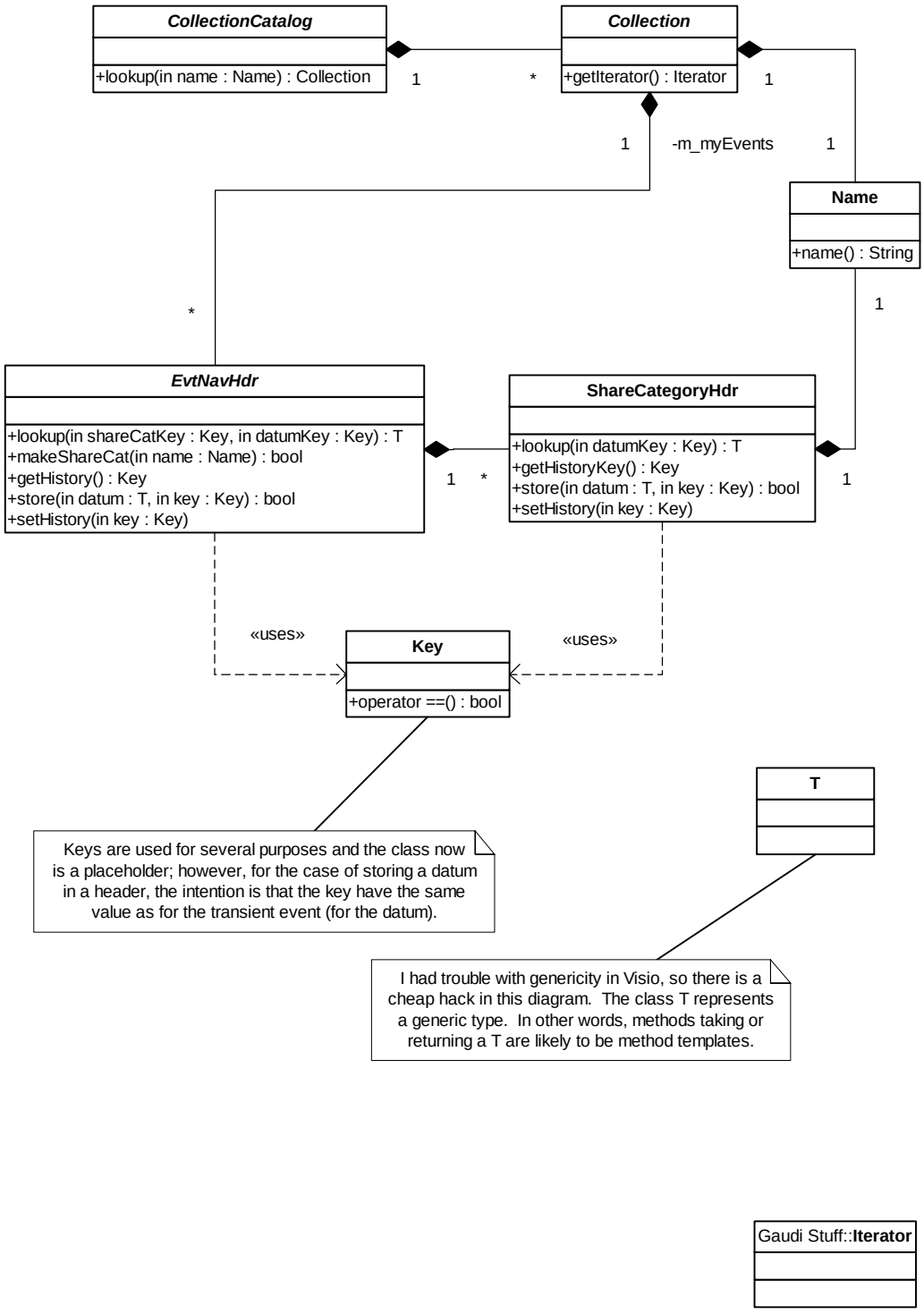

Figure 5: Data Structural Representation

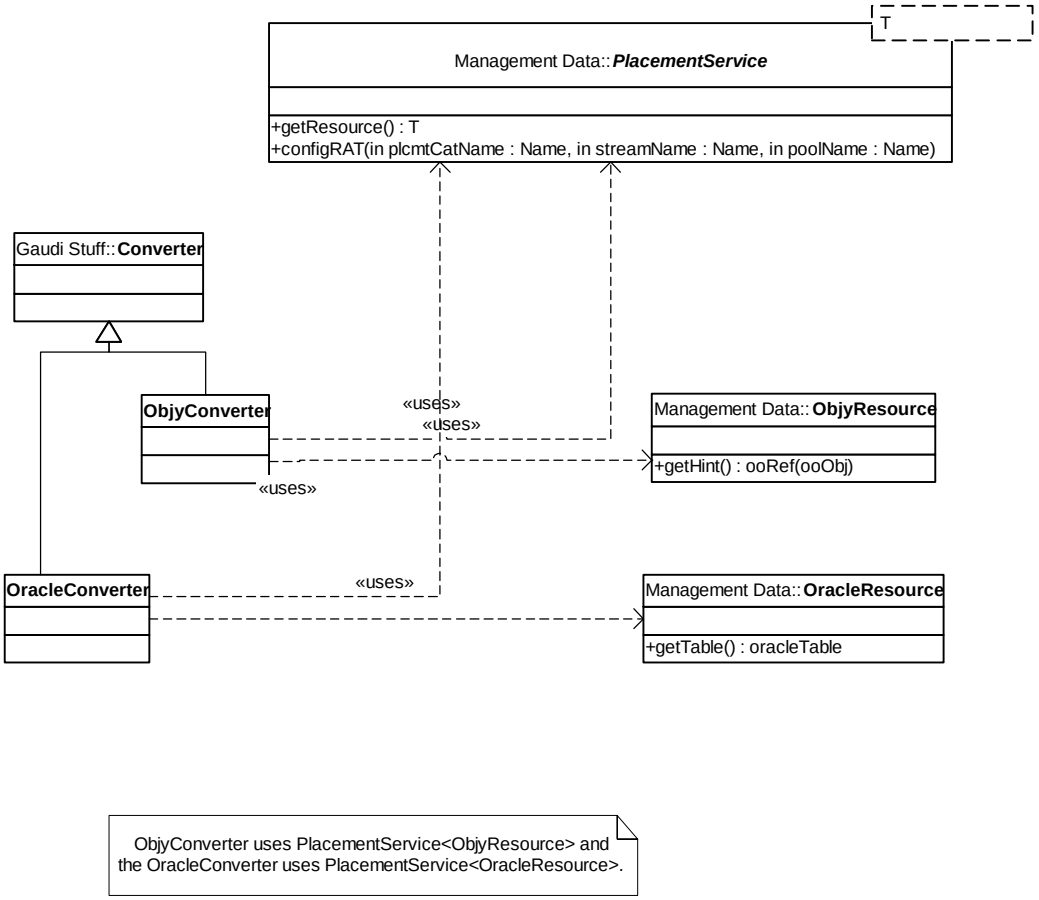

Figure 6: Athena Interface Representation

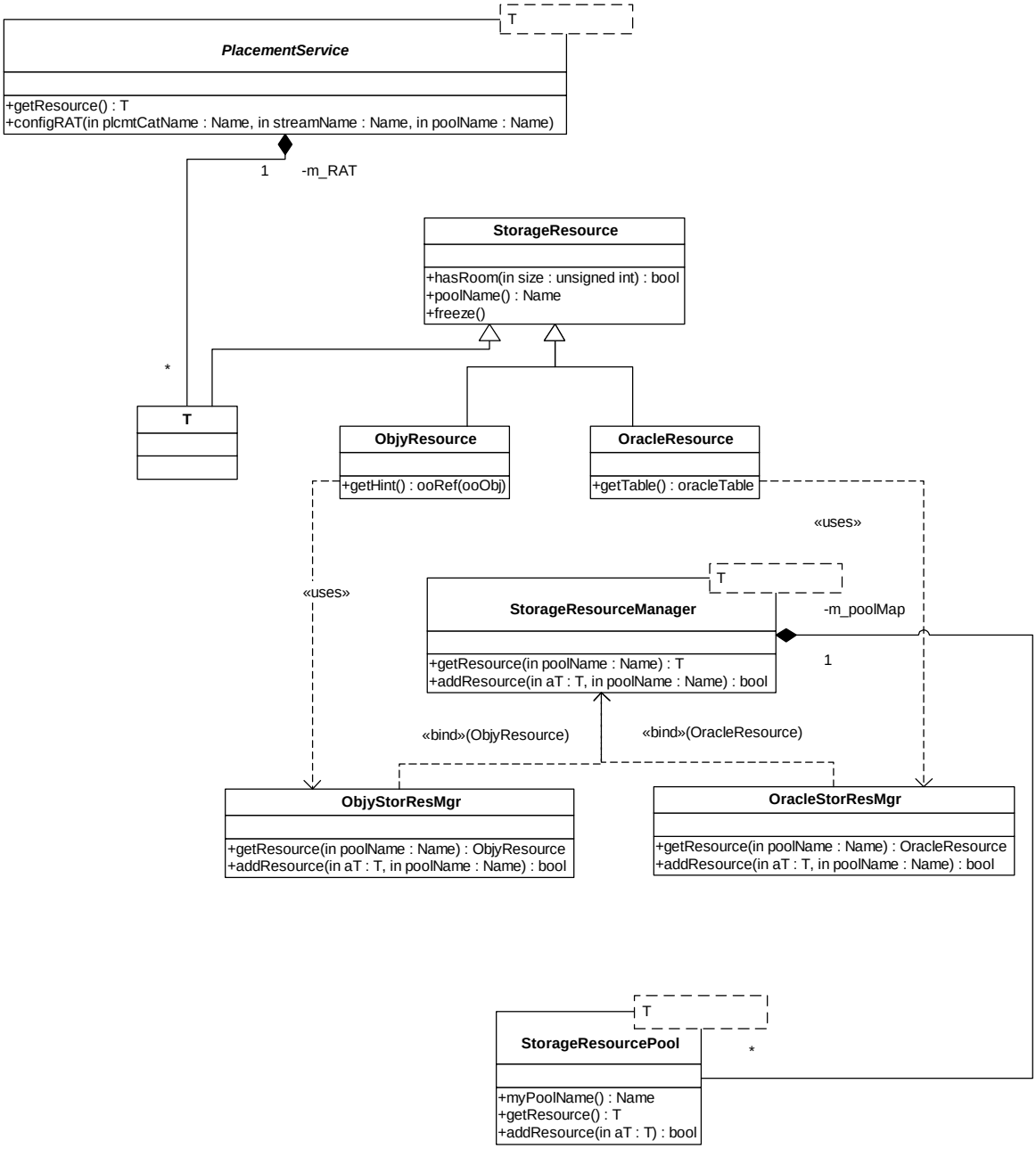

Figure 7: Management Data Representation

2. Glossary

- Stream
  A set of events selected by some criterion during production
- Collection
  An index of events.
- Management data

Sometimes called meta data, but we avoid that term as it is vague. Management data include lists of used files, associations between collections and files, etc.

- Persistent Event
  Provides navigation to persistent objects.
- Sharing Category

- TBD
  To be distributed
- Placement Category
- Writing Stream
- Non-Writing Stream
- DST
  Data Summary Tape – Traditionally, full events were summarized and abridged. In ATLAS this role will be served by the Event Summary Data (ESD).
- Query Engine
- **RAT**
  Resource Allocation Table (See Section 3). The RAT associates storage resources with placement categories on an output stream by output stream basis.

3. Bibliography

[1] E. D. Frank, "ATLAS Database Requirements Analysis," The University of Chicago, Enrico Fermi Institute and The Computation Institute 2001.

[2] P. Mato and ATLAS Collaboration, "GAUDI LHCb Data Processing Applications - Framework - Architecture Design Document," CERN - European Organization for Nuclear Research, Geneva, CH 1998.

[3] ATLAS Collaboration, "ATLAS Computing Technical Proposal," CERN - European Organization for Nuclear Research, Geneva, CH CERN/LHCC 96-43, 15 December 1996.